Levity Polymorphism in Haskell

…and other things that aren't discussed very often

Joe Kachmar

16 June 2020

Berlin Functional Programming Group

Preamble

- The Glasgow Haskell Compiler (GHC) is *very advanced*
    - It can (and will) aggressively optimize high-level code
    - Assume *all* code that follows is subject to optimization
    - What’s true for `-O0` is not necessarily true for `-O2`

Preamble

- The Glasgow Haskell Compiler (GHC) is *very advanced*
  - It can (and will) aggressively optimize high-level code
  - Assume *all* code that follows is subject to optimization
  - What's true for `-O0` is not necessarily true for `-O2`
- Much of what follows is being actively researched
  - Things may change subtly between releases of GHC

Preamble

- The Glasgow Haskell Compiler (GHC) is *very advanced*
  - It can (and will) aggressively optimize high-level code
  - Assume *all* code that follows is subject to optimization
  - What's true for `-O0` is not necessarily true for `-O2`
- Much of what follows is being actively researched
  - Things may change subtly between releases of GHC
- I'm **not** an active user of all of these features
  - Trust, but verify; I'll likely have gotten some of this wrong

Haskell, what is it good for?

Haskell, what is it good for?

Haskell is known for being:

- pure
- functional
- statically typed

Haskell, what is it good for?

Haskell is known for being:

- pure
- functional
- statically typed

- **lazy**
- **high-level**
- **higher-kinded**

Haskell, what is it good for?

Haskell is **not** known for:

- “obvious” space/time complexity
  - consequence of lazy evaluation
- precise control over memory allocation
  - not uncommon in “high-level”[1] languages

[1] “High-level” and “low-level” are fairly squishy terms; for the sake of example consider Python to be high-level and C to be low-level

Laziness, Lifted Types, and Levity Polymorphism

Laziness

```
data Boolean = False | True
```

Laziness

```haskell
data Boolean = False | True
```

`False` is a *value* whose *type* is `Boolean`

```
λ> :type False
False :: Boolean
```

Laziness

```haskell
data Boolean = False | True
```

`False` is a *value* whose *type* is `Boolean`

```
λ> :type False
False :: Boolean
```

`Boolean` is a *type* whose *kind* is `Type`

```
λ> :kind Boolean
Boolean :: Type
```

Laziness

```haskell
data Boolean = False | True
```

`False` is a *value* whose *type* is `Boolean`

```
λ> :type False
False :: Boolean
```

`Boolean` is a *type* whose *kind* is `Type`

```
λ> :kind Boolean
Boolean :: Type
```

Kinds are like "types of types"

Laziness

```
example :: [Boolean]
example = [True, error "boom!"]
```

Laziness

```haskell
example :: [Boolean]
example = [True, error "boom!"]
```

```
λ> head example
True
```

`example` is a lazy value

- `error` isn't evaluated on definition
- `head` doesn't touch `error`

Laziness

```
example :: [Boolean]
example = [True, error "boom!"]
```

```
λ> head example
True
```

`example` is a lazy value

- `error` isn't evaluated on definition
- `head` doesn't touch `error`

```
λ> head (tail example)
*** Exception: boom!
```

Laziness and Lifted Types

What we wrote:

```
data Boolean = True | False
```

What we believed:

`Boolean` values may to be one of...

- `True`
- `False`

Laziness and Lifted Types

What we actually got:

```haskell
data Boolean = True | False | ⊥
```

`Boolean` values may *actually* be one of...

- `True`
- `False`
- `⊥` or “bottom”
    - Computations which never complete successfully [2]
    - Frequently a result of `undefined` or `error`

[2] https://wiki.haskell.org/Bottom

Laziness, Lifted Types, and Levity Polymorphism

What *is* a `Type`, anyway?

Laziness, Lifted Types, and Levity Polymorphism

What *is* a `Type`, anyway?

```
data TYPE (a :: RuntimeRep)
data RuntimeRep = LiftedRep | UnliftedRep | Int8Rep | ...
type Type = (TYPE 'LiftedRep)
```

- `TYPE` is the abstract “kind” of valid Haskell types
  - Parameterized by runtime representation
- `TYPE LiftedRep` is a “lifted type”
  - **Lazy** types
  - “Normal” Haskell `Types`
- `TYPE UnliftedRep` is an “unlifted type”
  - **Strict** types
- `TYPE IntRep` is a “primitive type”
  - Strict 8-bit signed integer types

Levity Polymorphism

What we think Haskell gives us:

```
($) :: (a → b) → a → b
f $ x = f x
```

Levity Polymorphism

What we think Haskell gives us:

```haskell
($) :: (a → b) → a → b
f $ x = f x
```

What Haskell *actually* gives us:

```haskell
($) :: forall r a (b :: TYPE r). (a → b) → a → b
f $ x = f x
```

- `f :: (a :: Type) → (b :: TYPE r)`
  - Accepts a lifted type
  - Returns a type that is *levity-polymorphic*
    - i.e. polymorphic over its “liftedness”
    - More precisely it is polymorphic over its *representation*

Levity Polymorphism

Levity polymorphism and unlifted types have restrictions...

- Levity-polymorphic values cannot be *bound*
  - `fn0 x = ...` or `let x = ...` are illegal when `x :: TYPE r`
- Unlifted types cannot be bound at the *top-level*
  - `fn1 :: (a :: TYPE 'UnliftedRep)` is illegal
- Error messages and type signatures can be confusing

Recap

- Haskell is a lazy language
- Lazy values are “lifted”
- Strict values are “unlifted”
- Levity polymorphism abstracts over this distinction

Questions?

Runtime Representation and Memory Allocation

Runtime Representation

[...] calling convention is an implementation-level scheme for how subroutines receive parameters from their caller and how they return a result. [3]

[3] https://en.wikipedia.org/wiki/Calling_convention

Runtime Representation

[...] calling convention is an implementation-level scheme for how subroutines receive parameters from their caller and how they return a result. [3]

```
data RuntimeRep
  = LiftedRep | UnliftedRep | Int8Rep
  | TupleRep [RuntimeRep] | SumRep [RuntimeRep]
  ...
```

`RuntimeRep` abstracts over *calling convention*

[3] https://en.wikipedia.org/wiki/Calling_convention

Memory Allocation

Lifted types *must be* boxed [4]

- Boxed values are represented by pointers to heap-allocated objects

[4] Again, all of this may be completely optimized away

Memory Allocation

Lifted types *must be* boxed [4]

- Boxed values are represented by pointers to heap-allocated objects

Unboxed values *must* be unlifted

- `Int#`: unboxed, unlifted machine-sized integer

[4] Again, all of this may be completely optimized away

Memory Allocation

Lifted types *must be* boxed [4]

- Boxed values are represented by pointers to heap-allocated objects

Unboxed values *must* be unlifted

- `Int#`: unboxed, unlifted machine-sized integer

Unlifted structures *may* contain lifted values

- `Array# Int`: boxed, unlifted array of lifted integers

[4] Again, all of this may be completely optimized away

Memory Allocation

Why care about this?

Memory Allocation

Why care about this?

Unboxed values come with some guarantees:

- Memory representation is *static* and stack-allocated
- Can be stored directly in register memory
- Can be *deterministically* made to be *very* efficient

Runtime Representation and Memory Allocation

```haskell
add_int :: Int# → Int# → Int#
add_int i1 i2 = i2 +# i3
```

- `i1`, `i2`, `i3` are **stack**-allocated machine-sized signed integers
- `+#` is a primop wrapper for native integer addition[5]

[5] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/prim-ops

Runtime Representation and Memory Allocation

```
add_int :: Int# -> Int# -> Int#
add_int i1 i2 = i1 +# i2
```

Figure 1: `add_int` Function Definition

```
Example_add_int_info:
        addq %rsi,%r14
        movq %r14,%rbx
        jmp *(%rbp)
```

Figure 2: `add_int` Assembly

Unboxed Tuples

```
(Int, Int)
```

- Pointer to a heap object also pointing to heap objects[6]

[6] Again, all of this may be completely optimized away

Unboxed Tuples

`(Int, Int)`

- Pointer to a heap object also pointing to heap objects[6]

`(# Int, Int #)`

- Unboxed tuple of lifted `Ints`
- Contiguously spaced pointers to heap-allocated `Ints`

[6] Again, all of this may be completely optimized away

Unboxed Tuples

`(Int, Int)`

- Pointer to a heap object also pointing to heap objects[6]

`(# Int, Int #)`

- Unboxed tuple of lifted `Int`s
- Contiguously spaced pointers to heap-allocated `Int`s

`(# Int#, Int# #)`

- Unboxed tuple of unboxed `Int#`s
- Two machine-sized integers in contiguous memory

[6] Again, all of this may be completely optimized away

Unboxed Sums

```
data IntOrFloat = Int64 | Double | Word64
```

- All pointer-indirected, as with the tuples

[7] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/heap-objects#info-tables

Unboxed Sums

```
data IntOrFloat = Int64 | Double | Word64
```

- All pointer-indirected, as with the tuples

```
type IntOrFloat# = (# Int64# | Double# | Word64# #)
```

- Three words on 64-bit architectures
  - Tag word identifying the constructor
  - Info table pointer[7]
  - Data word (containing the actual data)

[7] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/heap-objects#info-tables
[8] https://gitlab.haskell.org/ghc/ghc/-/wikis/unpacked-sum-types

Unboxed Sums

```haskell
data IntOrFloat = Int64 | Double | Word64
```

- All pointer-indirected, as with the tuples

```haskell
type IntOrFloat# = (# Int64# | Double# | Word64# #)
```

- Three words on 64-bit architectures
  - Tag word identifying the constructor
  - Info table pointer[7]
  - Data word (containing the actual data)

GHC *should* optimize the former down to the latter[8]

[7] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/heap-objects#info-tables
[8] https://gitlab.haskell.org/ghc/ghc/-/wikis/unpacked-sum-types

Low-Overhead Abstractions in Haskell

```haskell
type Maybe# a = (# a | (##) #)

pattern Just# :: a -> Maybe# a
pattern Just# a = (# a | #)

pattern Nothing# :: Maybe# a
pattern Nothing# = (# | (##) #)
```

- `(##)`: empty, unboxed tuple
- Pattern synonyms to aid construction
- GHC 8.10's `UnliftedNewtypes` makes this easier

Miscellany

```
{-# language MagicHash, UnboxedSums, UnboxedTuples #-}
```

`MagicHash`: `#` may be used postfix in names

- `Int64#`: type constructor for unboxed 64-bit integers
- `I64#`: data constructor for 64-bit integers
  - Has the type `Int# → Int64`

```
import GHC.Exts
```

`GHC.Exts`: provides primitive functionality

- “Approved” re-exports from `GHC.Prim` module[9]

---

[9] https://hackage.haskell.org/package/ghc-prim-0.6.1/docs/GHC-Prim.html

Related Reading

`url-bytes`: URL parser

- Demonstrates some of the present ergonomic issues

`parsnip`: ANSI string parser combinators

Unlifted Data Types Wiki Entry

- Unlifted Data Types GHC Proposal

Unarisation GHC Source Code

- Explanation of Unarisation (by chessai)

Questions?